Exp-PIFO: Scalable and Efficient Programmable Packet Scheduling

HABIB MOSTAFAEI, Eindhoven University of Technology, The Netherlands
MOHAMMAD EZZATI, Technische Universität Berlin, Germany
PIETER J. L. CUIJPERS, Eindhoven University of Technology & Radboud Universiteit, The Netherlands
STEFAN SCHMID, Technische Universität Berlin & Fraunhofer SIT, Germany
GÁBOR RÉTVÁRI, Budapest University of Technology and Economics, Hungary
SEM BORST, Eindhoven University of Technology, The Netherlands

Programmable packet schedulers provide great flexibility in the ordering of packet transmission. They allow network operators to optimize crucial performance metrics, such as Flow Completion Time (FCT), using strategies that adapt to changes in the characteristics of incoming traffic. A challenge in programming these devices is that they are severely limited in registers, memory, and control flow operations. The popular scheduling strategy Push-In First-Out (PIFO), for example, cannot be straightforwardly implemented because it relies on sorting packets, which is difficult to implement at line speed on current hardware. Fixed-priority approximations of PIFO, like SP-PIFO and AIFO, do have hardware implementations but generally still do not scale well in, for example, the number of memory cells being used.

This paper introduces a new PIFO approximation strategy, Exp-PIFO, which prioritizes packets based on adaptive exponential prioritization criteria, called exponential bins. Exp-PIFO approximates the behavior of PIFO using only two memory cells to keep track of its state and uses a lookup table to avoid a complex control flow. We initially expected our improvement in memory and computation to come at a cost w.r.t. FCT performance compared to PIFO and its existing approximations. However, our empirical evaluation shows that Exp-PIFO sometimes even outperforms strict PIFO. We provide an explanation for this behavior and demonstrate the practical feasibility of Exp-PIFO through a proof-of-concept implementation on an Intel Tofino switch, which uses significantly less memory than comparable implementations of SP-PIFO and AIFO.

CCS Concepts: • **Networks** → **Programmable networks**; **Network performance analysis**.

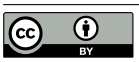
Additional Key Words and Phrases: Programmable packet scheduler, Programmable networks, exponential binning

ACM Reference Format:
Habib Mostafaei, Mohammad Ezzati, Pieter J. L. Cuijpers, Stefan Schmid, Gábor Rétvári, and Sem Borst. 2025. Exp-PIFO: Scalable and Efficient Programmable Packet Scheduling. *Proc. ACM Netw.* 3, CoNEXT3, Article 17 (September 2025), 20 pages. https://doi.org/10.1145/3749217

1 Introduction

Programmable networks for optimizing traffic flows. Given the increasingly stringent performance requirements on communication networks, the networking community has recently put great efforts into designing more flexible and programmable networks, enabling a wide range

Authors' Contact Information: Habib Mostafaei, Eindhoven University of Technology, Eindhoven, The Netherlands; Mohammad Ezzati, Technische Universität Berlin, Berlin, Germany; Pieter J. L. Cuijpers, Eindhoven University of Technology & Radboud Universiteit, Eindhoven, The Netherlands; Stefan Schmid, Technische Universität Berlin & Fraunhofer SIT, Berlin, Germany; Gábor Rétvári, Budapest University of Technology and Economics, Budapest, Hungary; Sem Borst, Eindhoven University of Technology, Eindhoven, The Netherlands.



ACM 2834-5509/2025/9-ART17
https://doi.org/10.1145/3749217

of optimizations [5, 6, 14, 21, 27, 31]. A particularly interesting approach to improving network performance is programmable packet scheduling. Traditionally, vendors pre-programmed packet scheduling algorithms, and flexibility was limited to configuring a few parameters [13], such as queue length. Recently, however, we have been witnessing the emergence of more and more powerful programming abstractions supporting tailored scheduling algorithms, at least for strict priority and fair queuing.

The introduction of programmable scheduling allows operators to deploy packet schedulers on hardware [27]. It enables network operators to enact specific scheduling policies by assigning a *rank* value to packets, indicating their respective *priority* with high resolution. Programmable schedulers, in turn, process these tagged packets, scheduling them based on their assigned rank orders. This approach provides a flexible and customizable framework for tailoring network scheduling policies to meet specific operational requirements. Although great insights on clever algorithms for assigning ranks to packets have been obtained [17, 19, 24, 26, 27], existing solutions come with limitations in terms of hardware requirements and performance.

A common performance metric for which network operators wish to optimize is the average flow completion time (FCT), where *flow* refers to the total amount of data a host wants to transmit. This flow is divided into *packets* that can be transmitted individually. The FCT for a host is then defined as the total time from transmission of the first packet until all packets of a flow are received [15]. If packets are dropped in the network due to queue overflows, they will be retransmitted by the host.

PIFO and its approximations. Push-In First-Out (PIFO) queues have been proposed to provide an abstraction to implement the scheduling policies by sorting packets at line rate on hardware [27]. PIFO keeps the packets in the queue sorted using the 'push' primitive, which allows arbitrary positioning of incoming packets, and drains the packets from the head. Upon arrival of a packet, PIFO calculates its rank and places it into the transmission queue sorted by that rank (see Figure 1). However, although PIFO theoretically obtains very good results current hardware is still too restricted to implement this strategy at line rate [20]. The main issue is that current programmable devices do not support sorting enqueued packets.

On current hardware, approximations to the PIFO behavior have been proposed, which also obtain good performance [5, 6, 16, 21, 25, 31, 32]. Most notably, SP-PIFO [5] approximates PIFO behavior by assigning packets of 'similar rank' to the same fixed priority in an otherwise classical fixed priority scheduler. Order errors are therefore restricted to occur only within the same priority queue. Within this approach of approximating PIFO, it is still an open question what the best classification of 'similar rank' is to improve the FCT. An alternative approximation, AIFO [31], for example, uses binning according to an estimation of quantiles in the rank distribution.

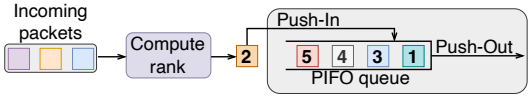


Fig. 1. Programmable packet scheduler with a PIFO queue.

Hardware limitations. Current programmable devices are limited in the number of registers available to the programmer, the total amount of memory, and the type of control flow operations. For the design of a successfully implementable PIFO approximation, which needs to leverage register read/write operations to compute equivalence classes of rank, and needs to keep track of a number of states to make the strategy adaptable, it is therefore important to keep these operations as simple and lightweight as possible, avoiding unnecessary use of registry operations, memory, and control flow decisions.

While current strategies like SP-PIFO are implementable within these restrictions, it turns out that they do not scale very well, for example, in the number of queues one would like to deploy. As a consequence, the resolution of the approximation cannot be controlled very well. By keeping our approximation strategy very light-weight, we manage to circumvent this problem for our proposed implementation.

Contribution of this paper. In this paper, we focus on improving the implementability of PIFO approximation by exploring exponential binning, i.e., using a binning strategy with increasing bin-sizes for lower priorities. This strategy turns out to be cheap in memory and computation. Moreover, it allows for dynamic adaptation to deal with variable flow size distributions by scaling the binning exponent. While we had expected this reduction in implementation complexity to come at the cost of an increase in FCT, exponential binning in fact turns out to outperform strict PIFO with respect to this metric in some realistic scenarios. We provide an explanation for this observation and discuss implications for the current approach for approximating PIFO, and reflect on whether approximation of PIFO is still the best basis when aiming to optimize the FCT.

In summary:

- We propose a simple scheduling mechanism using exponentially increasing bin sizes (base 2), which maps packet ranks to priority queues with minimal memory and computation. The approach adapts dynamically to changing traffic using a single runtime statistic.
- We provide a Tofino implementation that uses only six pipeline stages and significantly less memory than SP-PIFO (5.42× less) and AIFO (2.49× less).
- Our large-scale simulations show that Exp-PIFO reduces flow completion time (FCT) by up to 22.24% compared to PIFO and 24.45% compared to SP-PIFO. Moreover, analysis of the simulations shows that Exp-PIFO provides a better approximation of PIFO than SP-PIFO regarding packet-drop behavior.
- We analyze the performance improvement over PIFO to explore in which scenarios it is not optimal w.r.t. FCT. This gives our research a new direction, in which approximation of PIFO is still interesting, but no longer our main objective.
- We show that Exp-PIFO can handle a larger number of queues than SP-PIFO, which is restricted to a few queues due to the lack of support by currently available programmable devices.
- To support reproducibility, we release our code and experiment artifacts as an online appendix [1].

Outline. In Section 2, we introduce Exp-PIFO and discuss its general workings, illustrated by discussing how it behaves in typical websearch [7], Hadoop [22], and datamining [18] workloads that are often used as benchmarks. Furthermore, we give a pseudocode specification of the algorithm suitable for programmable packet schedulers and show how we encode and implement the bin sizes in P4 on a Tofino switch. In Section 3, we evaluate the performance of Exp-PIFO on the websearch and datamining workloads by comparing it to the performance of PIFO and SP-PIFO. We identify conditions under which PIFO may not be optimal w.r.t. FCT, and subsequently look into the packet-drop behavior of Exp-PIFO compared to that of SP-PIFO. Section 4 discusses some earlier work on programmable packet scheduling and other types of packet schedulers that inspired this paper. We conclude our work in Section 5.

[1] https://github.com/mostafaei/EXP-PIFO

2 The Exp-PIFO Algorithm

2.1 Adaptive Mapping of Rank to Priority

We consider a switch in which each egress port has M priority queues Q_j ($1 \leq j \leq M$), with Q_1 indicating the highest priority queue. The switch enqueues a series of packets p_i of different flows in this set of priority queues, and the goal is to assign priorities to packets so that the traffic through the switch is optimized for FCT.

A general approach to this is to assign higher priorities to packets originating from smaller flow sizes, similar to the shortest-processing-time first approach in queuing theory. However, as the number of queues in an egress port is limited, this requires a mapping from flow sizes to priorities. Because this method of mapping into priorities is, in fact, not specific to flow sizes, we utilize a mapping from a generic ranking parameter to priorities. In this paper, we simply speak of a rank, indicating some generic notion of importance, but we will keep the FCT as a motivating example in the back of our minds. We use r_i to indicate the rank of a packet p_i.

Moreover, as the distribution of flows in a network typically changes over time in real networks, our objective extends beyond identifying a static rank-to-priority mapping. We aim to devise a mapping that is not static but adaptive, adjusting to the evolving distribution of ranks observed in recent traffic while still optimizing the FCT.

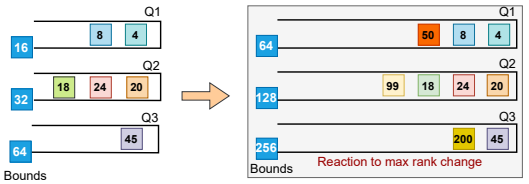


Fig. 2. Initial enqueuing of packets, and its update, according to the adaptive rank-to-queue function resulting from the Exp-PIFO algorithm, for 3 priority queues.

Figure 2 illustrates the adaptive rank-to-queue mapping that we propose in this paper and will explain in detail in the coming sections. Each colored box in Figure 2 indicates a packet, and the number within each box shows its rank. Notably, a packet with the lowest rank value signifies its association with a high-priority flow. Initially, Exp-PIFO observes six packets with a maximum rank of 45. It selects an exponent value β slightly higher than this maximum, and uses 2^β to set the upper bound of the lowest-priority queue (left side of Figure 2). As new traffic arrives, the queue bounds adapt–illustrated by the arrival of a packet with rank 200, which prompts an update to the queue boundaries (right side of Figure 2).

2.2 Exponential Mapping and Periodic Adaptation

Our approach, which we name Exp-PIFO, is based on a mapping from ranking to priority by assigning a bound b_j to each priority queue Q_j, where $j \in 1 \ldots M$ and M is the total number of queues. This bound serves as a threshold value that determines the admission of incoming packets. A packet of rank r would go into queue Q_1 if $0 \leq r \leq b_1$ and into Q_{j+1} if $b_j < r \leq b_{j+1}$. Higher ranks map to higher queue values, which we recall means lower priority.

As our primary aim in this paper is to achieve resource efficiency, and as we would like to be robust against large variations in flow size over time, our most important design choice is to use exponentially increasing bins. As a base formula, we choose $b_j = 2^{\beta \frac{j-1}{M-1} + \gamma}$. The shape of this formula is chosen such that γ purely determines the bound set for b_1 and $\beta + \gamma$ determines the bound of b_M. In our algorithm, γ is taken to be constant. As we aim to optimize the FCT, scheduling below the packet level is likely not relevant. The smallest relevant rank r_{min} is therefore equal to the maximum size of an Ethernet packet. To pick $b_1 = r_{min}$, we choose $\gamma = log_2(r_{min})$. Over time, we keep track of the highest recently observed rank r_{max} and, ideally, would similarly match its size to the lowest priority $b_M = r_{max}$ bin by choosing $\beta = log_2(r_{max}) - \gamma$.

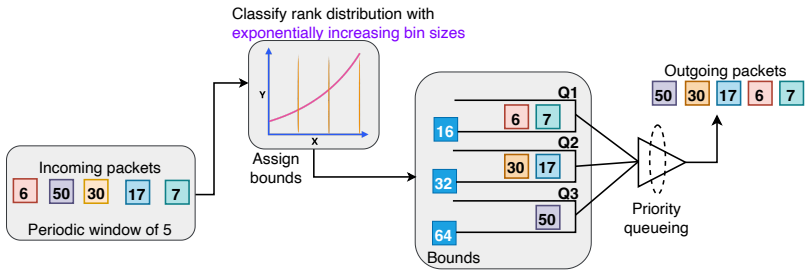


Fig. 3. Rank distribution identification and queue bounds assignment of Exp-PIFO. There are already some enqueued packets in each queue with a different color. The numbers are the rank indicators.

However, as we aim for low resource usage in hardware implementations, we have to think of β, which is a variable in our strategy, as an integer value. Implementing floating-point numbers on current programmable devices is infeasible. We consider two options for rounding β. Taking $\beta = \lceil log_2(r_{max}) - \gamma \rceil$ leads to an over-approximation of the bound on Q_M, causing a waste of queue-space while $\beta = \max(0, \lfloor log_2(r_{max}) - \gamma \rfloor)$ leads to an under-approximation, causing some packets to not being stored in Q_M at all. This latter problem is however easily solved by defining b_j only for $1 \leq j < M$ and effectively storing anything larger than b_{M-1} in b_M.

As initial experiments show that the performance difference when rounding up or down is small, and because the second option is less wasteful in terms of queue space than the first, we focus on rounding down for our calculations on β for the remainder of this paper.

Finally, to keep track of β throughout a periodically replenished window of size C, we implement a counter c that keeps track of the number of arrived packets. In other words, our replenishment strategy is not based on timing, but on inflow. Whenever a packet arrives with a rank r such that $\lfloor log_2(r) - \gamma \rfloor > \beta$, we increase the value of β to match this new maximum immediately. Whenever c reaches the maximum value C, we reset c to 0 and reset β to match the rank of the most recent packet, thus recalibrating the system to adapt to changing flows. Figure 3 illustrates the components that implement this approach.

Illustrative example. The main idea behind Exp-PIFO is that we save on computation and memory cost by using a single value to characterize our binning strategy, rather than needing one or more registers per queue. Let us illustrate our use of this adaptive exponential classifier using three real traffic distributions.

Figure 4(a) shows the Cumulative Distribution Function (CDF) of the flow size for websearch [7] datamining [18], and Hadoop [22] workloads. We set the simulation time to 10× the FCT of the largest flow and plotted the CDFs using the most recent 50% of observed rank values to reflect long-term behavior under steady-state conditions. As one can see, these distributions are quite heavy-tailed, with sometimes up to 20% of the flow sizes being much larger than the median flow size. This effect is aggravated if we look at the individual distribution of ranks in Figure 4(b,c,d). The packet rank is based on remaining flow size, meaning that large flows will contain both packets with a high rank (at the start of the flow) and with a low rank (at the end of the flow), while small flows contain only packets with a low rank. However, note that the ranks are depicted on a logarithmic scale, which shows that large flows contribute many more packets with a relatively high rank than with a relatively low rank. As we aim to capture the different distributions of Figure 4(b,c,d) with a single characteristic, we need at least an exponential function as a first-order approximation to this.

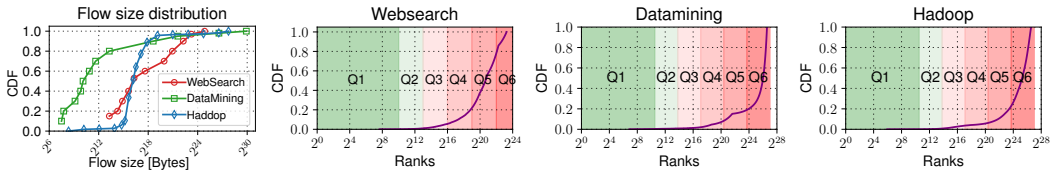


Fig. 4. Flow size distribution (a) Three typical examples of flow-size distributions in a datacenter setting (b) for websearch, (b) datamining, and (c) Hadoop workloads in a scenario with six priority queues.

The colored areas illustrate the bins that are chosen based on an exponential that is scaled between r_{min} and r_{max} in these figures. It is clear that this does not give an equiprobable distribution, but it does lead to a distribution that favors flows that have a high probability of finishing soon and is therefore expected to work reasonably well as an approximation of PIFO.

2.3 The Exp-PIFO algorithm

Algorithm 1 shows the pseudo-code of Exp-PIFO. The system starts with an initialization of the packet counter c and exponent β, which are kept as registers throughout the operation of the ingress control flow. Subsequently, the algorithm waits for arrival of a packet, reads the rank of the packet into a register, computes the floor of the logarithm of the rank, and compares this to the bound parameter β. If larger, the parameter β needs to be increased to ensure the queue bounds fit the current traffic. Subsequently, the algorithm increases the packet counter and checks whether the end of the current window has been reached. If so, the counter and the bound parameter are re-initialized. Finally, the correct queue is computed from the logarithm of the rank, and the packet is forwarded to this queue.

Clearly, the least straightforward operations in this algorithm are the computation of the logarithm of the rank in the function *compute_log*(x) which returns $\max(0, \lfloor log_2(x) - \gamma \rfloor)$ in line 6 and the arithmetic computing the queuing index, *compute_queue_id*(x, β) which returns $((x + 1)/(M - 1)) \div \beta + 1$, in line 13. All other operations are straightforward

Algorithm 1: Ingress control using Exp-PIFO.
Inputs: window size C and number of queues M.

```
1  c = 0 ▷ Init register for packet counter
2  β = 0 ▷ Init register for exponent
3  while true do
4      wait_for_packet_arrival()
5      x := obtain_packet_rank() ▷ x = r
6      x := compute_log(x) ▷ x := max(0, ⌊log2(r) − γ⌋)
7      if x > β then
8          β := x ▷ Update exponent
9      c = c + 1 ▷ Inc packet counter
10     if c > C then
11         c := 0 ▷ Start new window
12         β := x ▷ Re-initialize exponent
13     x := compute_queue_id(x, β)
         ▷ x := ((x + 1) · (M − 1) ÷ β) + 1
14   ▷ Now:
       x = ((max(0, ⌊log2(r) − γ⌋) + 1) · (M − 1)) ÷ β + 1
15   ▷ assuming M > 1 and x > 1 we derive
16   ▷ firstly: b_{x−1} = 2^{β (x−2)/(M−1) + γ} ≤ r
17   ▷ and secondly: r < 2^{β (x−1)/(M−1) + γ} = b_x
18     if x ≤ M then
19         enqueue_packet_in(x)
20     else
21         enqueue_packet_in(M)
```

comparisons, additions, or memory operations. In the next section, we will discuss how to implement these functions on specific hardware, Tofino, using P4 as a programming language.

The exponentially increasing bins are what distinguishes Exp-PIFO from previous works [5, 25] that try to approximate the behavior of PIFO queues without the need for packet ordering. Crucially, we observe that this choice allows us to represent the current rank to priority mapping using only a single memory cell β, while allowing for a large range in possible ranks. Regardless of whether flows have typical sizes of KBs or MBs, the algorithm can scale its workings to match the current distribution.

Furthermore, Exp-PIFO uses periodic updates of its mapping to adapt to changes in the traffic distribution. Notably, approximation mistakes will briefly be higher after each periodic update because the rank to priority mapping is initially reset, but during our experiments, we have found that these moments are generally brief and do not influence the performance significantly w.r.t. optimizing the FCT. Depending on the use-case, one could consider an additional memory cell that keeps track of the maximum value of β found in the previous period, and use this upon re-initialization. This is however left as a topic for future research.

In summary, our initial design of Exp-PIFO achieves its design goal of low memory footprint and good scalability and adaptability using only two memory cells for tracking the rank of recent packets and the number of arrived packets in the most recent update window. Next, we will illustrate the effectiveness of our proposed approach, and discuss how it can be implemented using limited registry and control flow operations.

2.4 Tofino Implementation

While implementing the algorithm outlined in the previous section on a Tofino switch [2] using the P4 [12] language, we have to keep a number of restrictions in mind. Firstly, while processing a packet on a Tofino switch, the number of read and write operations per register per packet is limited. We do have metadata at our disposition to perform computations or store a read register value, but those variables are valid as long as the packet resides within the switch and memory is freed when a packet leaves the switch. Secondly, the number of registers available is limited, but our algorithm has been designed to only use two. Thirdly, as there is limited support by the data plane in doing complex arithmetic, we aim to carefully use the precious memory of the device through look-up tables. Therefore, we have to keep the sizes and number of input variables in mind while designing those look-up tables.

To get an impression of the range of rank values that we need to be able to process, and hence the number of bits required for our registers, we study the values that occur in a number of typical traffic workloads. Figure 4(a) shows the flow size distribution of websearch [7], datamining [18], and Hadoop [22] workloads. Additionally, considering a synthetic Pareto trace [11], we can observe that the exponent values (i.e., the value of $\lfloor log_2(r_i) \rfloor$) of a packet p_i can vary in a range of 6 to 33. Some of these flow distributions are more skewed, such as Pareto. We can also notice that the exponent value varies between 7 and 30 in the datamining workload, while this variation is limited on the websearch. This means that the variable x in the algorithm in Algorithm 1 has to be able to carry at least 33 bits, but also that 6 bits would suffice for the register β that carries only the exponent value.

The look-up table for $compute\_log(x)$ in line 6 in Algorithm 1 can, in principle, be implemented as a look-up table of range match-kind. Given 33 possible consecutive outputs and 33 input bits, we would need $33 \times 33 = 1.089$ bits of memory.

Alternatively, it is also possible to split the input and first read the 17 most-significant bits of the rank r_i into a variable x_h and subsequently the 16-bit least-significant bits into x_l. Mathematically, $x = 2^{16} \cdot x_h + x_l$ so $\lfloor log_2(x) \rfloor = 16 + \lfloor log_2(x_h) \rfloor$ if $x_h > 0$ and $\lfloor log_2(x) \rfloor = \lfloor log_2(x_l) \rfloor$ if $x_h = 0$.

Consequently, a $17 \times 17 = 289$ bit look-up table can be called on both the most- and least-significant bits, and the result can be combined into the appropriate exponent using a comparison and addition. Depending on how much space the coding of this approach takes, the memory footprint of it may even be smaller than the first option. Notably, using this approach, the complexity of the algorithm scales logarithmically with a range of ranking values.

The look-up table for $compute\_queue\_id(x, \beta)$ in line 13 of our algorithm likely needs to be match-by-value as it takes two input values. Both inputs are in the range of 6 bits, giving a 12-bit input value in total. The computation produces one output value representing the queue identifier Q_j. As most switches support 8-32 queues, we can represent this output in 5 bits. The look-up table itself can, therefore, be fit into $2^{12} \times 5 = 20.480$ bits, or 2.5 KByte. Note that, in fact, this look-up table can also immediately implement the comparison carried out in lines 18-21, thus simplifying the algorithm somewhat in one go.

In comparison, one should note that other PIFO approximations, like SP-PIFO, need to keep each of their queue bounds in registers to provide for the adaptability that is required. As a consequence, these approaches require more memory, and adapting the bounds to support a larger number of queues becomes challenging. Current implementations of SP-PIFO are limited to the use of at most 8 queues. By characterizing the entire binning using a single parameter, we avoid much of these problems.

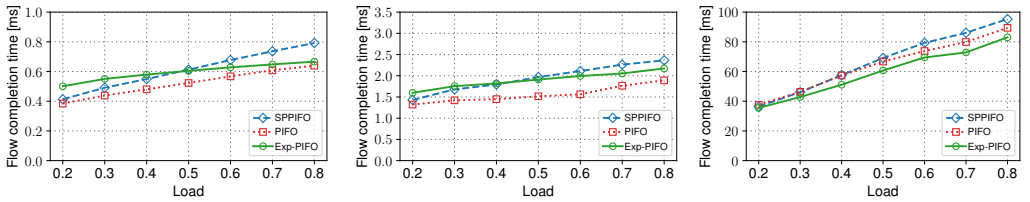
3 Evaluation

This section first reports the performance of Exp-PIFO using packet-level simulations in different terms of performance metrics. Then, we evaluate the resource efficiency of Exp-PIFO implementation on Tofino switches and compare it with SP-PIFO [5] and AIFO [31]. Finally, we report the results of running experiments on our testbed with a Tofino switch.

3.1 Packet-level Simulations

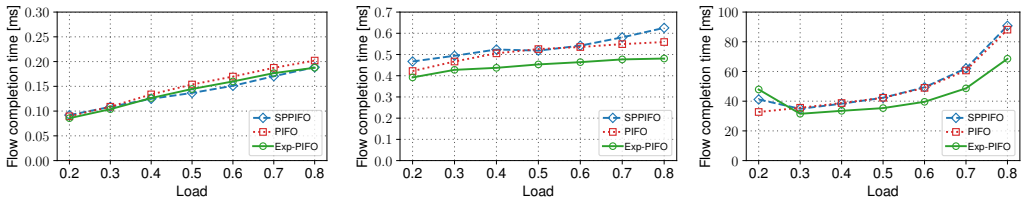
The main goal of designing Exp-PIFO is to simplify the implementation of programmable packet schedulers and improve resource efficiency; however, we are also concerned with its performance, particularly in reducing FCTs of flows. For this reason, we implemented a packet-level simulator.

Methodology. We use packet-level simulations to study the performance of Exp-PIFO in various terms. We implement Exp-PIFO in Netbench [1] and use a leaf-spine datacenter topology with four spine switches, nine leaf switches, and 144 servers. We configure the same link settings of SP-PIFO for the topology by adjusting the bandwidth of access links to 1Gbps and the leaf-spine links to 10Gbps. Our simulation implementation computes the rank of incoming packets based on the remaining flow size, following the Shortest Remaining Processing Time (SRPT) policy used in pFabric [8]. Consistent with PIFO and its approximations, our implementation does not explicitly incorporate starvation prevention mechanisms. However, prior work, such as PDA [29] (and others), has proposed solutions to address starvation, which can also be integrated with Exp-PIFO if needed. As discussed in [8], pFabric rate control approximation is done using the standard TCP implementation with a retransmission time of $3 \times$ RTTs to balance the differences in TCP Retransmission Timeout (RTOs) among proportional queue sizes. Additionally, we configure the queue size to accommodate 80 packets across all experiments. In the case of PIFO, all 80 packets are allocated to a single queue. For SP-PIFO and Exp-PIFO, the 80 packets are evenly divided across 8 queues, with each queue assigned a size of 10 packets. We empirically generate traffic using two real-world workloads: websearch [7] and datamining [18]. Both traffic workloads are heavy-tailed and have packet arrival time following a Poisson distribution. We route the packets using ECMP and select a



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 5. The FCT of flows across different loads on websearch workload.



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 6. The FCT of flows across different loads on datamining workloads.

random pair of source-destination servers.

Evaluation objective. The main performance objective is the FCT, but we also evaluate the performance of Exp-PIFO for throughput and fairness. We compare the performance of Exp-PIFO with those of PIFO [27] and SP-PIFO [5]. We ignore reporting the results for TCP and DCTCP since their performance results have already been reported in the previous works, e.g., SP-PIFO. We also check the impact of the different number of queues on the performance of Exp-PIFO. We use eight strict priority queues for the evaluation unless we specify another value, and the periodic window size is 5000 packets. To obtain a more intuitive presentation of our results, we classify the flows into two categories: 1) small flows when the size is $<$ 100KB and 2) large flows when $\geq$ 1MB.

3.1.1 Improvement over PIFO. Figure 5 shows that Exp-PIFO can reduce the FCT of large flow sizes up to 10.9% and 15.35% on websearch compared with PIFO and SP-PIFO. This reduction in FCT is up to 22.24% and 24.45% in datamining workload (see Figure 6). It is remarkable that Exp-PIFO outperforms our target strategy PIFO in these scenarios. Furthermore, it is interesting to see that this improvement is particularly notable under high network load conditions. The improvement is greater for the datamining workload than for the websearch workload. Next, we first address the reasons why PIFO is not an optimal scheduling strategy for FCT in general, and subsequently discuss how Exp-PIFO is able to exploit this.

PIFO in the context of trickle flows. The application of PIFO to FCT was inspired by the Shortest-Remaining-Processing-Time-First (SRPT) policy in scheduling theory, which is proven to be optimal in terms of overall average FCT when the flow size is known upon arrival. The idea is to use PIFO with a rank set equal to the *remaining* flow size of a packet, so that shorter remaining flows obtain higher priority, thus leading to a presumably optimal FCT. However, the proof of optimality of the SRPT scheduling policy implicitly assumes a scenario with infinite buffer sizes and

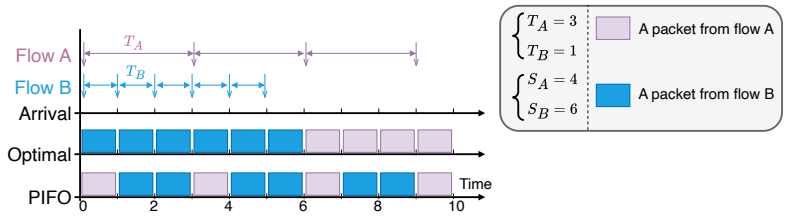
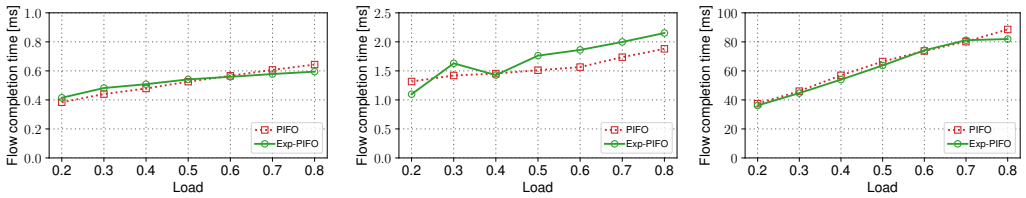


Fig. 7. An example scenario to show PIFO and optimal scheduler behavior in the presence of trickle flows.



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

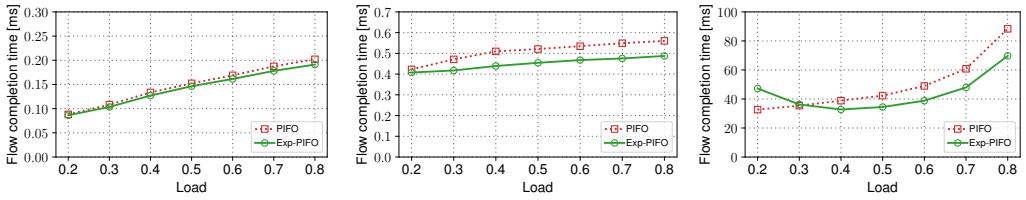
Fig. 8. The FCT of flows across different loads on websearch workload with a large queue size.

the entire workload of a flow arriving as a single instantaneous burst. As it turns out, in the absence of these conditions, there are scenarios in which the SRPT scheduling policy is non-optimal.

As an example, consider the scenario in Figure 7, in which packets of two flows need to be scheduled. All packets in all flows have the same unit size. Flow A has a size of S_A packets with a fixed inter-arrival time of T_A, while flow B has a size of S_B packets with an inter-arrival time of T_B. Both flows start at the same time. We note that the earliest finish time of a flow X if given full capacity would be $(S_X - 1) \cdot T_X + 1$ in this case.

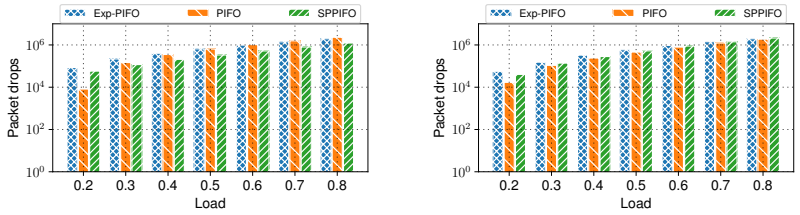
Next, suppose furthermore that A represents a small trickle flow, while B represents a large burst flow. Comparatively, this means that we assume $S_B > S_A$ (the flow B is larger than A) while $(S_B - 1) \cdot T_B + 1 < (S_A - 1) * T_A + 1$ (if given full capacity, B can still finish before A because of trickling). In that scenario, the optimal FCT is achieved by actually giving the large burst B priority. Any interference by a packet from the trickle flow A would lower the FCT. PIFO, however, in this scenario chooses to prioritize at least the first few packets from flow A, until sufficiently many from B are processed in the intermediate space to let the remaining flow size drop below that of A. Hence, PIFO is not FCT optimal in such cases.

Trickle flows in varying load and multi-hop scenarios. The question we address next is how such trickle flows can arise in the scenarios of Figures 5 and 6. It was observed above, that Exp-PIFO obtains its best performance w.r.t. PIFO on high workloads, which leads us to conjecture that packet drops may cause trickle flow behavior. In the case of the original PIFO strategy, packets with the largest remaining flow size are dropped when the queue is full. In the case of SP-PIFO and Exp-PIFO, the decision to drop a packet takes place after a queue has been assigned to a packet. If that particular queue is found to be full, the packet is dropped. As a consequence, depending on the queueing strategy, a system under high load may tend to drop high-priority packets instead of preferring to drop low-priority packets. Packet drops lead to retransmissions after a time-out, which causes them to be spaced in time.



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 9. The FCT of flows across different loads on datamining workloads with a large queue size.



(a) websearch (b) datamining

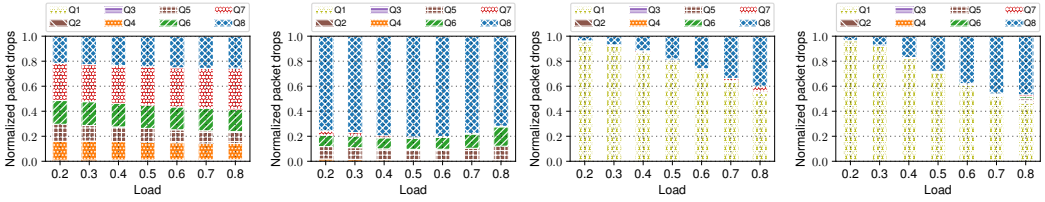
Fig. 10. The simulation results for packet drops of packet schedulers on websearch and datamining workloads.

To verify whether packet drops are indeed the main explanation of the improvement of Exp-PIFO over PIFO, we run the same simulation scenarios using very large buffers. The results of this in Figures 8 and 9 show that, indeed, for small and average loads, the performance of Exp-PIFO and PIFO almost coincide. However, for large flows, the situation has hardly changed. Apparently, where packet drops can significantly delay a smaller flow, the effect on larger flows is not sufficient to explain the performance increase.

Reflecting on possible causes further, we note that, in a multi-hop network like the one used in the datamining scenario, trickling may also be caused by a different mechanism. If, for example, in one hop a larger flow A gets frequently interrupted by newly arriving smaller flows, this may result in a trickle of A further downstream. If it merges with an even larger flow B, we obtain the scenario explained in the previous section. However, testing this hypothesis requires a new methodology, which we leave for future work.

3.1.2 Comparison of packet drops in PIFO approximations. What still remains relevant is the question of how well Exp-PIFO approximates PIFO in terms of packet drops. The sub-optimality example sketched above and the large buffer experiment both suggest that creating trickles in small flows (i.e., dropping packets of low rank) leads to worse FCT performance than creating trickles in larger flows. The decision of PIFO to drop packets of larger rank should, therefore, still be approximated as closely as possible. As we show next, Exp-PIFO is closer to PIFO than SP-PIFO in this respect.

Figure 10 shows the total number of dropped packets by PIFO, SP-PIFO, and Exp-PIFO for websearch and datamining workloads. We can observe that Exp-PIFO drops slightly more packets than PIFO and SP-PIFO on the websearch workload when the load is less than 40%. Both Exp-PIFO and SP-PIFO drop very similar numbers of packets when the link load is higher than 40%.



(a) websearch (Exp-PIFO) (b) Datamining (Exp-PIFO) (c) websearch (SP-PIFO) (d) Datamining (SP-PIFO)

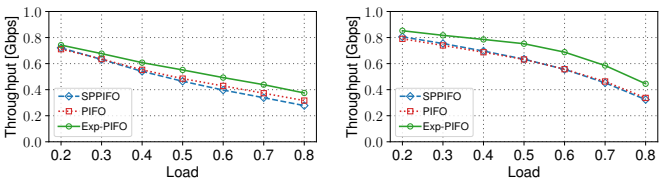
Fig. 11. The number of packet drops per queue in Exp-PIFO and SP-PIFO for various loads on websearch and datamining workloads.

Furthermore, we see a completely different packet drop behavior on the datamining workload in Figure 10(b). Exp-PIFO drops more packets when the load is low, but close to SP-PIFO and PIFO when the load increases. We investigate this difference further by looking into the queues from which these packets are dropped.

Figure 11 shows the relative number of packet drops from each queue, for both Exp-PIFO and SP-PIFO, in both scenarios. Figure 11(a) shows that, in Exp-PIFO, around 50% of the packet drops on the websearch workload are from the last two queues, i.e., from the lowest priorities, while Figure 11(c) we see that between 50% and 90% of the packet drops comes from the first queue in SP-PIFO, i.e. from high priority packets. This behavior of SP-PIFO is in stark contrast with the decisions taken in PIFO. Similarly, Figure 11(b) indicates that, for Exp-PIFO, around 75% of the packet drops occur from the two lowest priority queues.

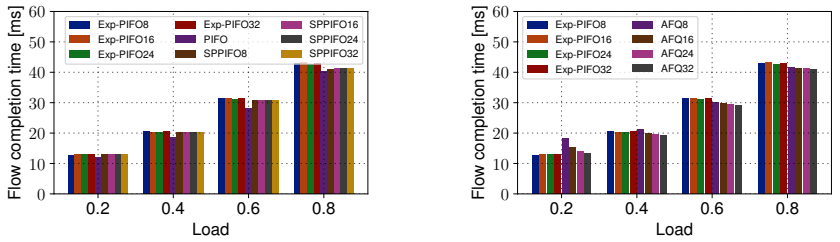
It becomes clear from this comparison that the packet drop behavior of Exp-PIFO is closer to that of the theoretical PIFO strategy than SP-PIFO. Note that this observation is independent of the choice of rank, so we may expect similar improvements in some non-FCT targeted settings. In any situation where the PIFO rank has been chosen to optimize a performance metric that is influenced by packet drops, a comparison like the one performed above may be relevant.

3.1.3 Throughput. This section reports the average throughput of large flows for websearch (see Figure 12(a)) and datamining workloads (see Figure 12(b)) . The large flows' throughput is the ratio between the total number of received bytes and the sum of the FCTs. All programmable schedulers achieve the highest throughput when the load is 0.2, with a drop in throughput as the load increases, regardless of the scheduling mechanism. This trend can be explained by the prioritization of small flows by the schedulers: as the load increases, the number of packets competing for priority grows, and the schedulers are more likely to drop packets with higher ranks, which are often from large flows. However, Exp-PIFO achieves the highest throughput among SP-PIFO and PIFO even at higher loads. Additionally, we observe that SP-PIFO and PIFO have similar throughput across both workloads as the load varies.



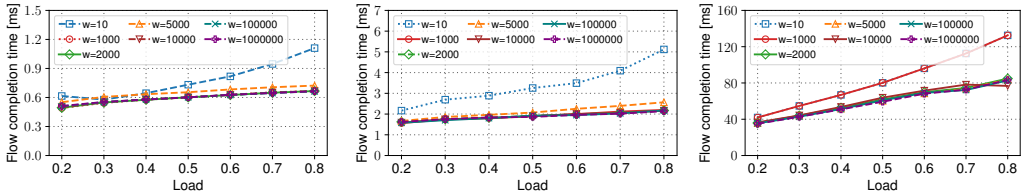
(a) Websearch (b) Datamining

Fig. 12. The simulation results for throughput of large flows of packet schedulers on websearch and datamining workloads.



(a) Programmable schedulers (b) AFQ

Fig. 13. The FCT results of websearch workload for fairness scenarios with a different number of queues. Fairness comparison with (a) programmable packet schedulers and (b) AFQ.

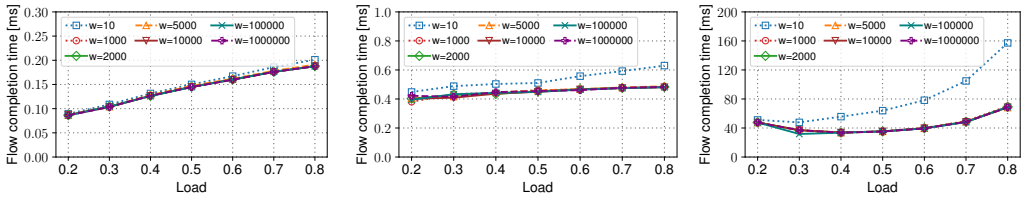


(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 14. The impact of the sliding window size of Exp-PIFO on the FCT of flows for websearch workload.

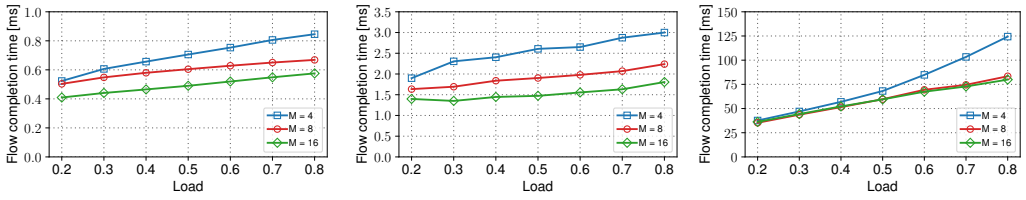
3.1.4 Fair Queueing. This section examines the fairness among the programmable packet schedulers. We achieve fairness among flows by replacing the SRPT-based rank assignment with a Start-Time Fair Queueing (STFQ) rank computation [17], implemented on top of PIFO, SP-PIFO, and Exp-PIFO. Specifically, the rank of each packet is calculated based on its virtual start time, which ensures fair bandwidth sharing across flows. To provide a broader perspective, we also benchmark our approach against Approximate Fair Queueing (AFQ). We evaluate fairness by analyzing FCTs – considering both the average and 99th percentile – across flows of different sizes (small, medium, and large) and under varying numbers of queues. This setup allows us to assess the effectiveness of each scheduling strategy in providing fair service under realistic datacenter workloads. Figure 13(a) shows the average FCTs of small flows for scenarios with a different number of queues for SP-PIFO and Exp-PIFO. Besides pushing the FCT of flows, Exp-PIFO gains a similar performance objective in fairness scenarios. Figure 13(b) indicates that Exp-PIFO can perform similarly to AFQ when the link load increases. However, when the link load is smaller, Exp-PIFO can further reduce the FCT of small flows than AFQ.

3.1.5 Impact of Window Size. We now evaluate the effect of periodic windows on the FCT of small and large flows in Exp-PIFO. Figure 14 and Figure 15 show that having a periodic window size of 1000 packets is enough to capture a good estimation from the distribution of ranks in the websearch and datamining workloads. To further understand the sensitivity of Exp-PIFO to the periodic window size, we reran the experiments with a broader range of values, including more extreme cases such as 10 and 1,000,000 packets. We find that, except for very small windows (e.g., 10 packets), where the system lacks sufficient data to accurately estimate queue bounds, Exp-PIFO



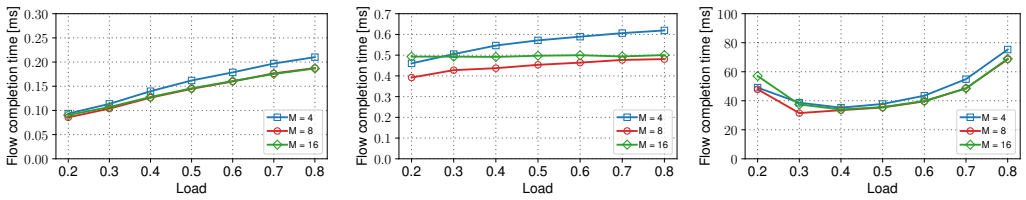
(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 15. The impact of the sliding window size of Exp-PIFO on the FCT of flows for datamining workload.



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 16. The effect of queue parameter on the FCT of flows for websearch workload.



(a) Average FCT for small flows (b) 99th percentile FCT for small flows (c) Average FCT for large flows

Fig. 17. The effect of queue parameter on the FCT of flows for datamining workload.

has similar performance for other window sizes, indicating robustness across a wide operational range. However, changing the periodic window size has little impact on the FCT of flows for both workloads. We observe a slight 99^{th} percentile FCT improvement when the window size is 5000 packets since Exp-PIFO can correctly adapt the queue bounds to improve the FCT. We also notice that when the load is 30% for the experiments using the datamining workload, the FCT of large flows is slightly increased when $w = 5000$. One reason for such behavior is that the system receives more large packets that end up in the wrong queue or are dropped by the admission mechanism of Exp-PIFO. Therefore, given its overall robustness and slight benefits at certain percentiles, we run the experiments using 5000 packets as the size of the periodic window.

3.1.6 Impact of Number of Queues. In this section, we report the FCT results of Exp-PIFO for scenarios when the number of strict priority queues M is set to 4, 8, and 16. Figure 16 and Figure 17 report the average FCT of small flows, the 99th percentile FCT of small flows, and the average FCT of large flows. We summarize the main findings as follows. (i) When the number of bins (queues) is small, the exponential mapping covers wide rank intervals per bin. This design compresses most

small flows into the highest-priority queues, ensuring they are quickly transmitted and largely isolated from larger flows–explaining their low tail latency. However, this coarse binning forces many large flows into a few low-priority queues, increasing buffer contention and exacerbating delays. These low-priority queues become bottlenecks, leading to worse average FCT for large flows. (ii) In contrast, increasing the number of queues creates finer-grained bins. Large flows are no longer grouped together, which spreads contention more evenly across queues and improves their throughput and latency. Yet, this added granularity comes with a cost: small flows are now distributed across more bins, weakening the strict prioritization they previously enjoyed. Consequently, small flows are more likely to be scheduled after marginally larger flows, increasing their tail latency.

3.2 Hardware Testbed

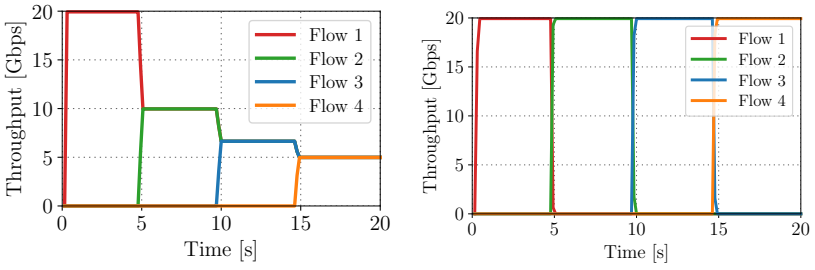
Resource consumption. Exp-PIFO is a resource-efficient programmable packet scheduler that leverages a small amount of the precious memory resources of the switches for enqueueing packets and leaves the rest for other purposes. For instance, operators need the memory of the switches to enable multi-tenancy. The saved memory by Exp-PIFO can be used to perform network measurements or machine learning tasks at the edge of the network.

Resource type	Exp-PIFO	SP-PIFO	AIFO
Match Crossbars	2.21%	8.05%	10.74%
Gateway	11.46%	10.62%	3.2%
Hash Bits	3.81%	4.81%	1.92%
SRAM	2.92%	18.75%	7.29%
TCAM	2.08%	0.42%	0%
Stateful ALUs	8.33%	20%	47.92%
Logical Table IDs	17.71%	18.12%	26.56%

Table 1. The resource consumption of Exp-PIFO, and reference implementations of AIFO and SP-PIFO on Intel Barefoot Tofino. The values are presented in percentages.

We first report on the resource efficiency of Exp-PIFO and compare it with the Tofino implementation of SP-PIFO and AIFO when implemented on Tofino *bf-sde-9.11.0*. AIFO uses one FIFO queue to schedule packets, while SP-PIFO and Exp-PIFO leverage multiple strict priority queues. Another key difference between Exp-PIFO and other packet schedulers is that they need to keep multiple states regarding the rank of incoming packets for admission. SP-PIFO leverages multiple registers to keep the queue bounds, while AIFO uses them to store the sampled packets for quantile computation. Table 2 indicates the resource consumption of Exp-PIFO, SP-PIFO, and AIFO. Despite using multiple priority queues for admission, Exp-PIFO needs 5.42× less SRAM and 2.4× less stateful ALUs than SP-PIFO. Exp-PIFO also needs 2.49× less SRAM and 5.75× less stateful ALUs compared with AIFO. We also report the resource consumption of Exp-PIFO when using multiple strict priority queues in Section A.1.

Bandwidth shares. We now report the performance of Exp-PIFO on our hardware testbed with a 3.2Tbps *Netberg Aurora 710* Tofino switch [4] connected to two servers via *Mellanox Connectx-5* through 100Gbps links. We use an 8-core CPU on the sender running an Intel(R) Core(TM) i7-6900K CPU @ 3.20GHz and on the receiver running an Intel(R) Xeon(R) w7-3465X. We develop both the



(a) FIFO (b) Exp-PIFO

Fig. 18. Bandwidth split of FIFO and Exp-PIFO for scenarios with four flows when R(Flow 1) < R(Flow 2) < R(Flow 3) < R(Flow 4).

sender and receiver using DPDK version 22.11.1 [3]. We run *Ubuntu 22.04.3 LTS* with the Linux kernel version *6.2.0-39-generic* on both servers. We throttle the capacity of the link connected from the switch to the receiver to 20Gbps.

We test the bandwidth split of FIFO and Exp-PIFO to four UDP flows of 20Gbps on a bottleneck link, each having different priorities similar to [5, 31, 32]. To demonstrate Exp-PIFO's prioritization capabilities, we intentionally throttle the link between the switch and the receiver to 20 Gbps, creating a controlled congestion scenario. Although the hardware supports 100 Gbps, reducing the link capacity allows us to simulate network contention and observe how Exp-PIFO prioritizes high-priority packets under constrained conditions. Without throttling, the effects of prioritization would be less apparent at the full 100 Gbps line rate. We start a flow every five seconds, and among the flows, the packets of *Flow 4* have the lowest rank (highest priority), while the packets of *Flow 1* have the highest rank. Figure 18 reports how FIFO and Exp-PIFO assign the bandwidth to different flows. FIFO evenly assigns the bandwidth on the arrival of a new flow every five seconds, ensuring fair sharing among active flows, as shown in Figure 18(a). Unlike FIFO, Exp-PIFO dynamically allocates the available bandwidth according to the rank of the four flows, with the highest-ranking flow dominating the link capacity. For example, during the second five-second interval, Exp-PIFO assigns the capacity of the bottleneck link to the packets of *Flow 2*, which have the highest rank, while the packets of *Flow 1* experience drops. We observed that SP-PIFO and AIFO achieved similar bandwidth shares in this experiment, and therefore, we omit the corresponding figures for brevity.

4 Related Work

Programmable packet scheduling. Packet scheduling has been widely studied by many researchers [19, 27]. The authors in [27] showed that programming the scheduling mechanisms could lead to the desired performance objective. However, the authors in [19] showed that a universal packet scheduling mechanism that aims at several performance objectives does not exist. Eiffel [23], Loom [28], and PIFO [27] have different approaches to aim for the desired goals of programmable schedulers. Eiffel [23] presents a queueing data structure that approximates the fine priority of each packet by leveraging the range of packet ranks using integer numbers. The design of the Eiffel simplifies its computational complexities but needs new hardware to employ its benefits. Loom [28] offers a new NIC design to offload the per-flow scheduling decisions from the OS to the NIC. Similar to Eiffel, Loom is not a general-purpose programmable device.

SP-PIFO approximates the PIFO behavior using multiple strict priority queues. However, while successful in concept, SP-PIFO has limited support from the current hardware in practice due to

its design dependence on per-queue registers for the approximation. Alternatively, AIFO [31] and RIFO [21] use a FIFO queue to admit packets based on their ranks. AIFO employs a quantile-based approach by sampling ranks and storing them in registers, but its scalability is limited by the number of registers available for quantile computation. In contrast, RIFO relies on min-max normalization to support in-queue ranking with constant-time operations, reducing memory and logic overhead in programmable schedulers. Packs [6] also relies on quantile computation to decide on packet admission, which inherits the same limitations of AIFO.

Alternative packet schedulers. PIAS [10] concentrates on identifying the distribution of the traffic flow sizes and uses multiple priority queues for packet admission, similar to pFabric [8]. pFabric maps packets to priority queues based on the size of the flow to which they belong. However, while pFabric relies on a priori traffic distribution information to make this mapping, PIAS [10] leverages Multi-level Feedback Queues (MLFQ) to achieve the Shortest Job First (SJF) goal. It accomplishes this by identifying the flow size distribution and dynamically transitioning flows from higher- to lower-priority queues as they transmit more bytes. Exp-PIFO has the same power to emulate queueing algorithms as PIFO, and when we happen to instantiate it with pFabric-like ranks, it seems to improve FCT for various traffic workloads.

Calendar queues [25] perform dynamic escalation of packet priorities to schedule packets using packet recirculation and control plane operations. Relying on the control plane to change the priority of every packet limits the throughput of calendar queues. Gearbox [16] and the work of [24] focus on the fair queueing aspects of programmable networks. Gearbox [16] utilizes a logical hierarchy of queuing levels to accommodate more packets using a few FIFO queues, allowing for discrepancies in the departure times of various packets. The generalization of Gearbox to approximate the PIFO queues has yet to be explored. Push-In-Extract-Out (PIEO) [26] introduced a new packet scheduling algorithm that allows for dequeueing packets from arbitrary positions. BBQ [9] and BMW-Tree [30] also provide a hardware implementation of PIFO queues. Nevertheless, PIEO, BBQ, and BMW-Tree are hardware designs that work on FPGAs, with different programmability and resource models compared to P4-based switches..

5 Conclusion

We presented Exp-PIFO, a simple and resource-efficient packet scheduler implementable on current available programmable switches with a small resource footprint. The design of Exp-PIFO needs just two memory cells to get insight into the rank distribution of incoming packets and enqueue them into the appropriate queues. Leveraging these two cells makes Exp-PIFO simple to implement and scale to support more priority queues. Furthermore, it pushes the flow completion time of programmable packet schedulers.

Exp-PIFO implements an approximation of Push-In-First-Out (PIFO) scheduling, but rather than focusing on improving the quality of approximation, our work focuses on implementability on resource-constrained devices. Nonetheless, simulation results show that the performance achieved by Exp-PIFO in terms of FCT is not only comparable to that of earlier approximations like SP-PIFO, but even shows an improvement over strict PIFO itself. This was unexpected to us, but can be explained from the fact that the optimality guarantees in terms of FCT for scheduling based on shortest-remaining-flow size do not apply in the presence of finite buffer sizes and trickle flows. Trickling may, for example, be caused by packet drops or downstream interference, which are often occurring phenomena in high-load multi-hop networks. Even though Exp-PIFO is not explicitly designed to exploit this fact, it appears to make favorable decisions in these cases. Furthermore, our results suggest that Exp-PIFO adapts well to changes in the flow characteristics over time. In our future work, we plan to study whether it is possible to structurally take trickling explicitly into account when scheduling flows, as we expect to be able to improve on the FCT even further.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG), SPP 2378 (project ReNO), 2023-2027. The work of Sem Borst was supported by The Netherlands Organisation for Scientific Research (NWO) through Gravitation grant NETWORKS 024.002.003.

References

[1] 2017. Netbench. https://github.com/ndal-eth/netbench.
[2] 2022. Intel Intelligent Fabric Processors. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.
[3] 2023. Data Plane Development Kit. https://dpdk.org.
[4] 2023. Netberg Aurora 710. https://netbergtw.com/products/aurora-710/.
[5] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 59–76.
[6] Albert Gran Alcoz, Balázs Vass, Pooria Namyar, Behnaz Arzani, Gabor Retvari, and Laurent Vanbever. 2025. Everything Matters in Programmable Packet Scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.
[7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. 63–74.
[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. PFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. 435–446.
[9] Nirav Atre, Hugo Sadok, and Justine Sherry. 2024. BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 455–475.
[10] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 455–468.
[11] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. 782–797.
[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[13] Mikkel Christiansen, Kevin Jeffay, David Ott, and F. Donelson Smith. 2000. Tuning RED for Web Traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 139–150.
[14] David D. Clark, Scott Shenker, and Lixia Zhang. 1992. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Conference Proceedings on Communications Architectures & Protocols* (Baltimore, Maryland, USA) *(SIGCOMM '92)*. 14–26.
[15] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.* 36, 1 (jan 2006), 59–62.
[16] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. 2022. Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 551–565.
[17] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Palo Alto, California, USA) *(SIGCOMM '96)*. 157–168.
[18] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (Barcelona, Spain) *(SIGCOMM '09)*. 51–62.
[19] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 501–521.
[20] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal Abstractions for Packet Scheduling. *Proc. ACM Program. Lang.* 7, Article 269 (2023).

[21] Habib Mostafaei, Maciej Pacut, and Stefan Schmid. 2025. RIFO: Pushing the Efficiency of Programmable Packet Schedulers. *IEEE Transactions on Networking* 33, 3 (2025), 1295–1308.

[22] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.

[23] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[24] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 1–16.

[25] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 685–699.

[26] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. 367–379.

[27] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 44–57.

[28] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and efficient {NIC} packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 33–46.

[29] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. 2022. QCluster: Clustering Packets for Flow Scheduling *(WWW '22)*. 1752–1763.

[30] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. 2023. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 208–219.

[31] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. 179–193.

[32] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. 2021. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 29–45.

A Appendix

A.1 Hardware Scalability with Queue Count

To evaluate the scalability of Exp-PIFO in hardware, we implemented configurations with 4, 8, 16, and 32 priority queues on Intel Barefoot Tofino using `sde-9.11.1`. Table 2 reports the percentage of each hardware resource used. Even with 32 queues, Exp-PIFO utilizes around 32% of any resource type, demonstrating that our design scales efficiently while remaining lightweight.

Table 2. Resource consumption of Exp-PIFO with varying numbers of queues on Intel Barefoot Tofino (shown as percentage of available hardware resources).

Resource type	4 queues	8 queues	16 queues	32 queues
Match Crossbars	2.21%	2.21%	2.01%	1.86%
Gateway	7.29%	11.46%	16.96%	27.34%
Hash Bits	3.81%	3.81%	3.21%	2.85%
SRAM	2.92%	2.92%	2.50%	2.19%
TCAM	2.08%	2.08%	1.79%	1.56%
Stateful ALUs	8.33%	8.33%	7.14%	6.25%
Logical Table IDs	13.54%	17.71%	22.32%	32.03%

Received December 2024; revised June 2025; accepted June 2025